STLMC Manual

Geunyeol Yu, Jia Lee, and Kyungmin Bae

Pohang University of Science and Technology, Pohang, Korea

1 Introduction

The STLMC tool is a robust model checker for signal temporal logic (STL) properties of hybrid system. STLMC can perform STL model checking up to a robustness threshold $\epsilon > 0$ for a wide range of nonlinear hybrid systems with ordinary differential equations.

The tool provides an expressive input format to easily specify a wide range of hybrid automata, and a visualization command to give an intuitive description of counterexamples. The tool is implemented in around 9,500 lines of Python code. It support various SMT solvers, such as Z3 [3], Yices2 [4], dReal [5] as an underlying solver to solve the robust STL model checking problem.

The architecture of the tool is illustrated in Figure 1. The tool takes a hybrid automaton H, an STL formula φ, a variable point bound N, time bound τ, and a threshold ϵ. The given STL formula φ is first translated into the ϵ-strengthening of the negated formula $\neg(\varphi^{+\epsilon})$. The SMT encoding $\Psi^{k,\tau}_{H,\neg(\varphi^{+\epsilon})}$ for $1 \leq k \leq N$ is then built using the STL bounded model checking algorithm [2, 8]. The satisfiability of $\Psi^{k,\tau}_{H,\neg(\varphi^{+\epsilon})}$ can be checked directly using an SMT solver or using the two-step optimization algorithm. If the tool find a counterexample, the tool provides visualization graphs of counterexample signals and robustness degrees.

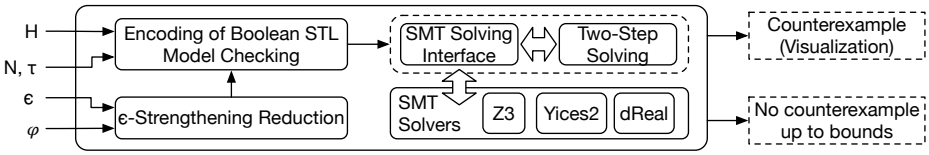

Fig. 1: The STLMC architecture

The tool is available at `https://stlmc.github.io/download` which explains how to download the tool.

The rest of the manual organized as follows. Section 2 describes a simple running example used to explain our tool. Section 3 explains the input model language of STLMC. Section 4 explains how to set analysis parameters for robust STL model checking using configuration files and command-line options. Finally, Section 5 shows how to visualize counterexample signals and robustness degrees.

2 Running example

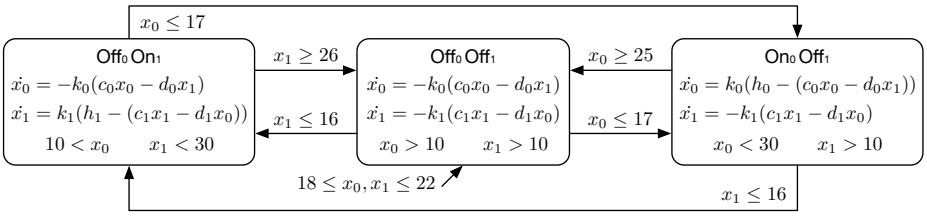

Fig. 2: A hybrid automaton for the networked thermostats.

We consider networked thermostat controllers adapted from [1,6]. There are two rooms connected by an open door. The temperature x_i of each room $i \in \{0, 1\}$ is controlled by each thermostat, depending on the heater's mode $q_i \in \{\mathsf{On}, \mathsf{Off}\}$ and the other room's temperature. The continuous dynamics of x_i can be given as ODEs as follows:

$$\dot{x_i} = \begin{cases} K_i(h_i - (c_i x_i - d_i x_{1-i})) & (\mathsf{On}) \\ -K_i(c_i x_i - d_i x_{1-i}) & (\mathsf{Off}), \end{cases}$$

where K_i, h_i, c_i, d_i are constants depending on the size of the room, the heater's power, and the size of the door. Fig. 2 shows a hybrid automaton of our thermostat controllers. Initially, both heaters are off and the temperatures are between 18 and 22. The jumps between modes then define a control logic to keep the temperatures within a certain range using only one heater.

For the input model in Fig. 3, (`therm.model`). the following command verified the formula `f1` up to bounds $N = 5$ and $\tau = 30$ with respect to robustness threshold $\epsilon = 1$ in 817 seconds using dReal:

```
$./stlmc ./therm.model -bound 5 -time-bound 30 -threshold 1 \
         -goal f1 -solver dreal -two-step -parallel
result : True (816.99192 seconds)
```

3 Modeling language

The STLmc tool supports model language to define hybrid system. The *model file* input format of STLmc, inspired by dReach [7], consists of five sections: variable declarations, mode definitions, initial conditions, state propositions, and STL properties. Mode definitions specifies flow, jump, and invariant conditions. Initial conditions specifies initial states of hybrid automata. STL formulas to be analyzed are defined in the STL properties section using proposition variables declared in state propositions. Fig. 3 shows the input model of the hybrid automaton described in the running example above.

```
const k0 = 0.015;        const k1 = 0.045;
const h0 = 100;          const h1 = 200;
const c0 = 0.98;         const c1 = 0.97;
const d0 = 0.01;         const d1 = 0.03;

int on0;                 int on1;
[10, 35] x0;             [10, 35] x1;

{ mode: on0 = 0;          on1 = 1;
  inv:  10 < x0; x1 < 30;
  flow: d/dt[x0] = - k0 * (c0 * x0 - d0 * x1);
        d/dt[x1] = k1 * (h1 - (c1 * x1 - d1 * x0));
  jump: x0 <= 17 => (and (on0' = 1) (on1' = 0)
                         (x0' = x0) (x1' = x1));
        x1 >= 26 => (and (on1' = 0) (on0' = on0)
                         (x0' = x0) (x1' = x1));
}
{ mode: on0 = 1;          on1 = 0;
  inv:  x0 < 30; x1 > 10;
  flow: d/dt[x0] = k0 * (h0 - (c0 * x0 - d0 * x1));
        d/dt[x1] = - k1 * (c1 * x1 - d1 * x0);
  jump: x1 <= 16 => (and (on0' = 0) (on1' = 1)
                         (x0' = x0) (x1' = x1));
    x0 >= 25 => (and (on0' = 0) (on1' = on1)
                     (x0' = x0) (x1' = x1));
}
{ mode: on0 = 0;          on1 = 0;
  inv:  x0 > 10; x1 > 10;
  flow: d/dt[x0] = - k0 * (c0 * x0 - d0 * x1);
        d/dt[x1] = - k1 * (c1 * x1 - d1 * x0);
  jump:
    x0 <= 17 => (and (on0' = 1) (on1' = on1)
                     (x0' = x0) (x1' = x1));
    x1 <= 16 => (and (on1' = 1) (on0' = on0)
                     (x0' = x0) (x1' = x1));
}

init: on0 = 0;  18 <= x0;  x0 <= 22;
       on1 = 0;  18 <= x1;  x1 <= 22;

proposition:
  [p1]: x0 - x1 >= 4;    [p2]: x0 - x1 <= -3;

goal:
  [f1]: <>[0, 3](x0 >= 13 U[0, inf) x1 <= 22);
  [f2]: [][2, 4](p1 -> <>[3, 10] p2);
```

Fig. 3: An input model example

3.1 Variable declarations

STLmc uses mode and continuous variables to specify discrete and continuous states of a hybrid automaton. *Mode variables* define a set of discrete modes Q, and *continuous variables* define a set of real-valued variables X.[1] We can also declare named constants whose values do not change.

Mode variables are declared with one of three types: `bool` variables with `true` and `false` values, `int` variables with integer values, and `real` variables with real values. E.g., the following declares a `bool` variable `b` and an `int` variable `i`:

```
bool b;    int i;
```

Continuous variables are declared with domain intervals. Domains can be any intervals of real numbers, including open, closed, and half-open intervals, E.g., the following declares two continuous variable `x` and `y`:

```
[0, 50] x;    (-1.1, 1) y;
```

Finally, constants are introduced with the `const` keyword. Constants can have Boolean, integer, or rational values. For example, the following declares a constant `k1` with a rational value 0.015:

```
const k1 = 0.015;
```

[1] An assignment to mode and continuous variables corresponds to a single state.

3.2 Mode definitions

In STLMC, *mode blocks* define mode, jump, invariant, and flow conditions for a group of modes in a hybrid automaton. Multiple model blocks can be declared, and each mode block consists of four components:

```
{
    mode: ...
    inv:  ...
    flow: ...
    jump: ...
}
```

A mode blocks represents a set of discrete modes of a hybrid automaton that satisfies conditions in the `mode` component. All states of the modes satisfy conditions in the `inv` component. States of the continuous variables in the modes evolves according to conditions in the `flow` component. A discrete transition can be possible if a state satisfies conditions in the `jump` component.

A `mode` component contains a semicolon-separated set of Boolean conditions over mode variables. The conjunction of these conditions represents a set of modes.[2] E.g., the following represents a set of two modes $\{(true, 1), (true, 2)\}$, provided there are two mode variables `b` (of `bool` type) and `i` (of `int` type):

```
b = true;    i > 0;    i < 3;
```

An `inv` component contains a semicolon-separated set of Boolean formulas over continuous variables. The conjunction of these conditions represents the invariant condition for each mode in the mode block. For example, the following declares an invariant condition for two continuous variables `x` and `y`:

```
x < 30;    y > - 0.5;
```

A `flow` component contains either a system of ordinary differential equations (ODEs) or a closed-form solution of ODEs. In STLMC, a system of ODEs over continuous variables $x_1, \ldots, x_n$ is written as a semicolon-separated equation of the following form, where e_i denotes an expression over $x_1, \ldots, x_n$:

```
d/dt[x_1] = e_1(x_1, ..., x_n) ;
...
d/dt[x_n] = e_n(x_1, ..., x_n) ;
```

For example, continuous dynamics of x and y that are specified as the ODEs: $\dot{x} = \sin(y)$ and $\dot{y} = y^2$ are defined as follows:

```
d/dt[x] = sin(y);  d/dt[y] = y ** 2;
```

[2] We assume that the mode conditions of different mode blocks cannot be satisfied at the same time, which can be automatically detected by our tool.

A closed-form solution of ODEs is written as a set of continuous functions, parameterized by a time variable `t` and the initial values $x_1(0), ..., x_n(0)$ for each discrete mode:

x_1`(t) =` e_1`(t,` $x_1(0)$`,...,`$x_n(0)$`) ;`
`...`
x_n`(t) =` e_n`(t,` $x_1(0)$`,...,`$x_n(0)$`) ;`

For example, continuous dynamics of x and y that are specified as the closed-form solutions: $x(t) = t^2 + 3 * t + x(0)$ and $y(t) = t + y(0)$ are defined as follows:

```
x(t) = t ** 2 + 3 * t + x(0);  y(t) = t + y(0);
```

A `jump` component contains a set of jump conditions *guard* `=>` *reset*, where *guard* and *reset* are Boolean conditions over mode and continuous variables. We use "primed" variables to denote states after jumps have occurred. E.g., the following defines a jump with four variables `b`, `i`, `x`, and `y` (declared above):

```
(and (i = 1) (x > 10)) => (and (b' = false) (i = 3) (x' = x) (y' = 0));
```

3.3 Initial conditions

In the `init` section, an initial condition is declared as a set of Boolean formulas over mode and continuous variables. Similarly, the conjunction of these conditions represents a set of initial modes. E.g., the following shows an initial condition with variables `b`, `i`, `x`, and `y`:

```
init: (and (not b) (i = 0) (19.9 <= x) (y = 0));
```

3.4 STL Properties

In the `goal` section, STL properties are declared with or without labels. In the case of a STL property declared with a label, the tool can only execute the formula by specifying the label.

For example, the following declares two STL formula in the running example. The first STL formula is declared without a label and the second STL formula is declared with label `f2`:

```
<>[0, 3](x0 >= 13 U[0, inf) x1 <= 22);
[f2]: [][2, 4]((x0 - x1 >= 4) -> <>[3, 10] (x0 - x1 <= -3));
```

To make it easy to write repeated propositions, "named" state propositions can be declared in the `proposition` section. For example, the second STL formula can be rewritten using two propositions `p1` and `p2` as follows:

```
proposition:
[p1]: x0 - x1 >= 4;
[p2]: x0 - x1 <= -3;

goal:
[f2]: [][2, 4](p1 -> <>[3, 10] p2);
```

4 Configuration

The STLmc tool can model check STL properties of hybrid automata, given three parameters $\epsilon > 0$ (robustness threshold), $\tau > 0$ (time bound), and $N \in \mathbb{N}$ (discrete bound). These analysis parameters for STLmc and options for underlying solvers can be defined in configuration files and command-line options. This section describes the parameters of the configurations and the hierarchy between the configuration file and the command line. We also explain how to run the STLmc tool using command-line interface.

4.1 Configuration files

Name	Explanation	Default
`bound`	a discrete bound $N \in \mathbb{N}$	-
`time-bound`	a time bound $\tau \in \mathbb{Q}^+$	-
`threshold`	a robustness threshold $\epsilon \in \mathbb{Q}^+$	0.01
`solver`	an SMT solver to be used (`z3`/`yices`/`dreal`)	`auto`
`time-horizon`	a mode duration bound	τ
`goal`	a list of STL goals to be analyzed	all
`two-step`	use the two-step optimization	disabled
`parallel`	parallelize the two-step optimization	disabled
`visualize`	generate extra visualization data	disabled
`verbose`	print more information of the execution	disabled

Table 1: STLmc configuration parameters.

A *configuration file* is defined in a .cfg file. The configuration file specifies the analysis parameters for STLmc and underlying solvers. Analysis parameters for STLmc are listed in Table 1. Fig. 4 shows a configuration example.

Analysis parameters for STLmc (Table 1) are defined within curly braces introduced with the `common` keyword. There are two mandatory parameters and eight optional parameters. When mandatory parameters are not set, STLmc throws an exception.

```
# STLmc configuration
common {
    # mandatory arguments
    # bound =
    # time-bound =

    threshold = 0.01            # positive rational number
    solver = auto               # dreal, yices, z3
    time-horizon = "time-bound"
    goal = "all"                # STL formula labels
    two-step = "false"          # on
    parallel = "false"          # on
    visualize = "false"         # on
    verbose = "false"           # print verbose messages
}

# underlying solver
z3      { logic = "QF_NRA" # QF_LRA }
yices   { logic = "QF_NRA" # QF_LRA }
dreal {
    precision = 0.001            # positive rational number
    ode-order = 5                # natural number
    ode-step = 0.001             # positive rational number
    executable-path = "../dreal" # dreal executable path
}
```

Fig. 4: A configuration example

Two parameters, `bound` and `time-bound`, are required parameters because the STLmc tool solves the robust STL model checking problem upto two bound parameters. A bound limits the number of mode changes and the number of *variable points*—at which the truth value of some STL subformula changes—in trajectories. A time bound bounds the time domain of trajectories. A threshold is a robustness threshold that bound the robustness degree of an STL formula. A time horizon T limits the maximum time duration of single modes in trajectories. If an STL formula is declared with a label, only that formula can be analyzed by providing the label as an argument to the `goal`. The parameters `two-step` and `parallel` enable the (parallelized) two-step optimization. When the `visualize` is set, extra data for visualization is generated.

We can choose different SMT solvers by setting an argument to the `solver`. The STLmc tool currently supports three SMT solvers: Z3 [3], Yices2 [4] and dReal [5]. The underlying solver is chosen depending on the flow conditions of a hybrid automaton. Z3 and Yices2 can deal with linear and polynomial functions, and dReal can deal with nonlinear ODEs—which is undecidable in general for hybrid automata—approximately up to a given precision $\delta > 0$.

Analysis parameters for each solver are defined within curly braces introduced with the *solver_ name* keyword. For `z3` and `yices2` there is only one parameter, `logic`, which sets the background logic for SMT solving. There are four parameters for the dreal3 solver. A `precision` sets δ for the framework of δ-complete decision procedures. A `ode-order` sets an maximum order of taylor expansion of ordinary differential equations. A `ode-step` sets a time step for numerical calculation. A `executable-path` defines a path to the dreal executable file.

For flexibility in setting analysis parameters, three levels of configurations are provided: (1) default configuration, (2) model configuration, and (3) model specification configuration. The model specific configuration inherits the model configuration, and the model configuration inherits the default configuration.

The STLmc tool provides a predefined configuration file, named default.cfg, in the top directory. The default configuration sets all parameters to their default values except for mandatory arguments (Refer Fig. 4)

The model configuration specifies some analysis parameters related to a specific model. For example, we want to analyze the input model in Fig. 2 at bound 5 and time bound 30 with respect to robustness threshold $\epsilon = 0.1$. Then, we only need to change the following three parameters: bound, time-bound, and threshold. Figure 5 shows the model configuration that specifies these partial parameters. Other parameters such as goal, parallel, etc, inherit the values defined in default.cfg.

Furthermore, people may want to run a model using a different configuration for each formula. People can instantiate a model specific configurations to instantiate some analysis parameters in a model configuration. For the same example above, we want to analyze a formula `f2` with respect to $\epsilon = 2$. Then, we only need to change the following two parameters: goal and threshold. Figure 6 shows the model specific configuration that specifies these partial parameters. Other parameters such as bound, time-bound, etc, inherit the values defined in a model configuration.

```
# STLmc configuration
common { bound = 5
         time-bound = 30
         threshold = 0.1 }
```

Fig. 5: A model configuration example

4.2 Command line options

The STLmc tool provides a command-line interface. The tool takes a model file, configuration files, and all parameters in a default configuration. The model file argument is required and other configuration arguments are optional.

```
# STLmc configuration
common { goal = f2
         threshold = 2 }
```

Fig. 6: A model specific configuration example

```
$./stlmc [path to model file] \
         -defaul-cfg [path to default config file]\
         -model-cfg [path to model config file] \
         -model-specific-cfg [path to model specific config file] \
         -bound [int] ...
```

If the `-default-cfg` option is not provided, the tool uses the predefined default configuration, named `default.cfg`, in the top directory. If the `-model-cfg` option is not given, the tool looks for a configuration file *<model_file>.cfg* in the path to model file. [3] If some parameters in the configuration are given as command-line options, those parameters override the values specified in the configuration files.

Example 1. Consider the input model in Fig. 3 (`therm.model`), the input model configuration in Fig. 5 (`them-model.cfg`), and the input model specification configuration in Fig. 6 (`them-model-spec.cfg`). The following command found a counterexample of the formula `f2` at bound 5 with respect to robustness threshold $\epsilon = 2$ in 8 seconds using dReal.

```
$./stlmc ./therm.model -model-cfg therm-model.cfg  \
         -model-specific-cfg therm-model-spec.cfg \
         -solver dreal -two-step -parallel -visualize
goal: []_[2.0,4.0] (p1 -> (<>_[3.0,10.0] p2))
result: counterexample found (bound 2)
running time: 7.46335 seconds
```

5 Visualization

The STLmc tool provides a script to visualize counterexamples for robust STL model checking. The visualization script takes a counterexample file and a visualization configuration file and returns graphs representing counterexample trajectories and robustness degrees.

```
./stlmc-vis [path to counterexample file] \
          -cfg [path to configuration file]
```

[3] If there is no matched model configuration file, the model configuration inherits the default configuration.

The counterexample file is specified in a .counterexample file. The file is created when there is a counterexample and the `visualize` option is set. The configuration file contains following things: (1) continuous variables, (2) STL subformula labels, (3) output format, and (4) group of variables. The file contains variable information of continuous variables and STL subformulas. The STLMC tool supports "html" and "pdf" as output formats. The states of variables in a group are plotted on one graph. Only variables of the same type can be grouped together. [4] Variables not assigned to a group are grouped together according to their type. Figure 7 shows a visualization configuration file of our thermostat controllers.

```
{   # continuous variables: x0, x1
    # STL subformula labels:
    # f2 --> [][2, 4]((x0 - x1 >= 4) -> <>[3,10](x0 - x1 <= -3))
    # f2_1 --> (x0 - x1 >= 4) -> <>[3,10](x0 - x1 <= -3)
    # f2_2 --> not (x0 - x1 >= 4)
    # f2_3 --> <>[3,10](x0 - x1 <= -3)
    # p_1 --> x0 - x1 >= 4
    # p_2 --> x0 - x1 <= -4

    # output = pdf # html
    group { (x0, x1), (f2, f2_1) (f2_2, f2_3) (p_1, p_2) }
```

Fig. 7: A visualization configuration example

Example 2. Consider the counterexample file for `f2` in Example 1 (`therm.model`) and the visualization configuration file in Fig. 7 (`therm_vis.cfg`). The following command generate a PDF image. in Fig. 8.

```
./stlmc-vis therm.counterexample -cfg therm_vis.cfg
```

The robustness degree of `f2` is less than ϵ at time 0, since the robustness degree of $\texttt{f2}_1$ goes below ϵ in the interval $[2, 4]$, which is because both the degrees of $\texttt{f2}_2$ and $\texttt{f2}_3$ are less than ϵ in $[2, 4]$. The robustness degree of $\texttt{f2}_3$ is less than ϵ in $[2, 4]$, since the robustness degree of $\texttt{p}_2$ is less than ϵ in $[5, 14] = [2, 4] + [3, 10]$.

[4] For example, if there is a group $(x0, f2)$, the tool will throw an error because the types of $x0$ and $f2$ are real and bool, respectively.

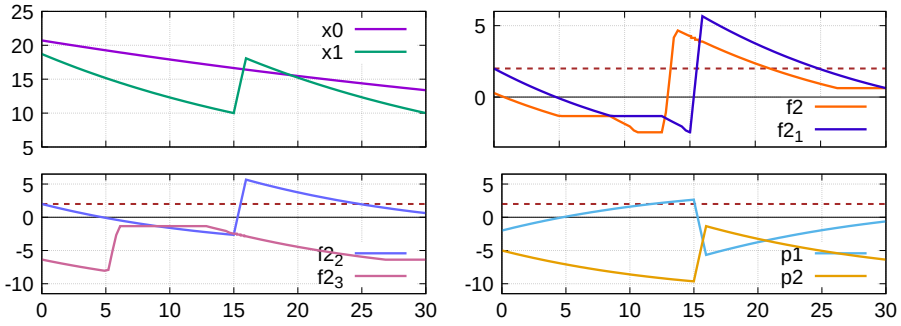

Fig. 8: Visualization of a counterexample (horizontal dotted lines denote $\epsilon = 2$).

References

1. Bae, K., Gao, S.: Modular smt-based analysis of nonlinear hybrid systems. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 180–187. FMCAD '17, FMCAD, Austin, TX (2017), http://dl.acm.org/citation.cfm?id=3168451.3168490
2. Bae, K., Lee, J.: Bounded model checking of signal temporal logic properties using syntactic separation. Proc. ACM Program. Lang. **3, POPL**(51), 1–30 (2019)
3. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proc. TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
4. Dutertre, B.: Yices 2.2. In: Biere, A., Bloem, R. (eds.) Proc. CAV. LNCS, vol. 8559, pp. 737–744. Springer (2014)
5. Gao, S., Kong, S., Clarke, E.M.: dReal: An SMT solver for nonlinear theories over the reals. In: Proc. CADE. LNCS, vol. 7898, pp. 208–214. Springer (2013)
6. Henzinger, T.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, NATO ASI Series, vol. 170, pp. 265–292. Springer (2000)
7. Kong, S., Gao, S., Chen, W., Clarke, E.M.: dReach: δ-reachability analysis for hybrid systems. In: Proc. TACAS. LNCS, vol. 7898, pp. 200–205. Springer (2015)
8. Lee, J., Yu, G., Bae, K.: Efficient smt-based model checking for signal temporal logic. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 343–354 (2021). https://doi.org/10.1109/ASE51524.2021.9678719